J. Symbolic Computation (1989) 7, 473–485

Backtrack Search with Isomorph Rejection and Consistency Check

C. W. H. LAM* AND L. THIEL

Computer Science Department, Concordia University, Montréal, Québec, Canada H3G 1M8

(*Received 7 September* 1987)

Performing an exhaustive backtrack search of combinatorial objects is always a dicey business. One can spend years of CPU time without finding anything. There is always the embarrassing question: "is the program correct?" The situation is even worse when the combinatorial object being searched for has a large symmetry group. For efficiency purposes, it is almost obligatory to use as much of the group as possible to prune isomorphic branches of the search tree. Yet, isomorph rejection is difficult to implement and is one of the major sources of errors. This paper proposes a new method of using the symmetry group, not only for isomorph rejection, but also as an independent consistency check of the correctness of the program. The number of isomorphic copies of a given solution is computed in two different ways and the results have to agree, which provides us with the independent consistency check.

1. Introduction

In this paper, we consider the problem of an exhaustive enumeration of combinatorial objects which are non-isomorphic under the action of a symmetry group. This problem is often solved using a backtracking method with isomorph rejection. The major difficulty in using this method is that large amounts of computer time and memory space are often required. However, the most embarrassing and often unanswered question is whether the implementation of the method is correct.

We are proposing a new method which uses the existence of a non-trivial symmetry group to derive some internal consistency checks. In particular, it has the following desirable properties.

1. It can handle several levels of isomorph rejection simultaneously and the consistency check is carried out at every appropriate level.
2. At each appropriate level, the orbit size of every partial solution is computed in two different ways, providing the consistency check.
3. It keeps the amount of space utilized small by deleting the record of a partial solution once the correct number of copies is seen.

The first descriptions of the backtracking method were in Walker (1960) and Golomb & Baumert (1965). There are now excellent descriptions in many standard textbooks such as Reingold *et al.* (1977) and Purdom & Brown (1985). The term isomorph rejection was introduced in Swift (1958). Our method is based on the existence of a general method for

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grants A9373, A9413, 0011 and by the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche under Grants EQ2369 and EQ3886.

0747–7171/89/050473+13 $03.00/0

isomorphism testing such as Butler & Lam (1985). The materials from group theory can be found in any standard textbook, such as Hall (1959). A restricted version of this consistency checking method has been used in the search for a projective plane of order 10 (Lam *et al.*, 1983, 1986).

In section 2, we present the basic ideas and definitions. In section 3, we give an outline of the isomorph rejection algorithm with consistency checks. Two examples are given in section 4. Section 5 gives a brief proof of the correctness of the consistency checking method as well as a few practical observations.

2. Basic Ideas

Before we can discuss the algorithm, we introduce some terminology. We define a search problem as follows:

> Given a collection of sets of candidates $C_1, C_2, C_3, \ldots, C_n$ and a boolean *predicate* function $\mathscr{P}$ defined on the Cartesian product $C_1 \times C_2 \times \cdots \times C_n$, find all n-tuples $(x_1, \ldots, x_n)$ such that $\mathscr{P}(x_1, \ldots, x_n)$ is true.

A brute-force approach would be to generate all the n-tuples and test each one to check whether it satisfies the predicate. Of course, this method would be too slow for most problems.

In a backtrack approach, we consider instead a boolean *partial predicate* $\mathscr{P}_k$ which is defined on all k-tuples with $k \leqslant n$ and whose value is the same as $\mathscr{P}$ when $k = n$. A k-tuple $(x_1, \ldots, x_k)$ is a *partial solution* at level k if the *partial* predicate $\mathscr{P}_k(x_1, \ldots, x_k)$ is true. The basic idea of the backtrack approach is to *extend* a partial solution at level k to one at level $k+1$, and if this extension is impossible, then to go back to the partial solution at level $k-1$ and attempt to generate a different partial solution at level k. However, we have to ensure that this process does not lead to a loss of solutions. The following *Consistency Criterion* ensures that the backtrack process is correct.

CONSISTENCY CRITERION. For all $k \leqslant n$, if the partial predicate $\mathscr{P}_k(x_1, \ldots, x_k)$ is true, then the partial predicates $\mathscr{P}_i(x_1, \ldots, x_i)$ are true for all $i < k$.

In other words, once a partial predicate $\mathscr{P}_i(x_1, \ldots, x_i)$ is false, then it is not necessary to extend it, because it will never lead to a complete solution.

Isomorph rejection depends on the existence of a property preserving symmetry group $\mathscr{S}(\mathscr{P})$ such that if $g \in \mathscr{S}(\mathscr{P})$ and g maps an n-tuple $(x_1, \ldots, x_n)$ to $(y_1, \ldots, y_n)$, then the values of the two predicates $\mathscr{P}(x_1, \ldots, x_n)$ and $\mathscr{P}(y_1, \ldots, y_n)$ are the same.

Let us first clarify the action of the symmetry group. We assume that $\mathbf{S}(\mathscr{P})$ acts on the n-tuples by permuting the indices. Since we are often working with subgroups of $\mathbf{S}(\mathscr{P})$ which permute the indices in a restricted manner, we introduce the notation $\langle \cdots | \cdots | \cdots \rangle$, where one or more of the symbol "|" separates the groups of consecutive indices which can permute amongst themselves. For example $\langle *^i | *^{n-i} \rangle$ is the subgroup of $\mathbf{S}(\mathscr{P})$ which permutes the first i indices and the remaining $(n-i)$ indices separately amongst themselves. Here, the special symbol '$*$' is used to denote candidates to be determined.

Using '$*$', a partial solution $(x_1, \ldots, x_k)$ can also be written as $(x_1, \ldots, x_k, *, \ldots, *)$ with $(n-k)$ $*$'s. Hence, the action of the symmetry group $\mathbf{S}(\mathscr{P})$ extends to the partial solutions. We assume that whenever two partial solutions are compared, a '$*$' always matches with a '$*$'.

Isomorph rejection is most useful if it can be applied to the intermediate levels of the search. The following is a typical situation. The candidates for the first few levels, say $(x, \ldots, x_i)$ are known and form a partial solution. It is extended to level $k \leqslant n$ and isomorph rejection is done. Implicitly, there exists a subgroup of $\mathbf{S}(\mathscr{P})$ which fixes $x_1, \ldots, x_i$ and permutes its extensions. Only non-isomorphic extensions, which are called *certificates*, under the action of this subgroup are considered further.

For simplicity, we assume that no initial candidates are given. If they do exist, we can use the original given partial solution $(x_1, \ldots, x_i)$ as context and define a new search problem of finding $(y_1, \ldots, y_{n-i})$ to complete the solution.

With this simplication, the group $\langle *^k | *^{n-k} \rangle$ permutes the partial solutions $(x_1, \ldots, x_k)$ amongst themselves. Isomorph rejection at this level means that only one representative from each orbit of $C_1 \times C_2 \times \cdots \times C_k$ under the action of $\langle *^k | *^{n-k} \rangle$ will be extended. Informally, we say that the subtree rooted at $(x_1, \ldots, x_k)$ is the same as the one rooted at $(y_1, \ldots, y_k)$ if there exists a g in $\langle *^k | *^{n-k} \rangle$ mapping $(x_1, \ldots, x_k)$ to $(y_1, \ldots, y_k)$.

We may not want to perform isomorph rejection at every level, because it may be too expensive to do or because there is insufficient information. We often perform isomorph rejection only at the top levels and then at selected intermediate levels. These levels are called *testing* levels in order to distinguish them from the level k of the partial solution $(x_1, \ldots, x_k)$, which is called a *normal* level. One has to remember, however, that the action of the symmetry group on the partial solutions is still best seen in terms of the normal levels. In terms of the testing levels, a candidate is a vector consisting of candidates at the normal levels. The symmetry group permutes the individual components of the vectors within and across the testing levels. Thus, the candidates themselves at the testing level may be changed by the action of the group.

The testing levels are labelled from 1 to N. The function α translates a testing level i to its normal level $\alpha(i)$. Thus the subgroup of $\mathbf{S}(\mathscr{P})$ acting on the partial solutions at the testing level i is $\langle *^{\alpha(i)} | *^{n-\alpha(i)} \rangle$, and is denoted by $\mathbf{S}_i(\mathscr{P})$.

There is still one difficulty. Consider two testing levels k and j with $k < j$. A $g \in \mathbf{S}_k(\mathscr{P})$ may not be an element in $\mathbf{S}_j(\mathscr{P})$, which makes it difficult to compute the orbit sizes of the rejected partial solutions at the testing level j. The example of graph enumeration in section 4 illustrates this behaviour. This difficulty implies that the consistency check may not work at every testing level. For this reason, a testing level j is said to be a *checking* level, if one can guarantee that every permutation g used in rejecting a partial solution $(x_1, \ldots, x_{\alpha(k)})$ at testing level k, $k \leqslant j$, is in $\mathbf{S}_j(\mathscr{P})$. These checking levels are special because there is a consistency check for every certificate at these levels. The first and final levels are always checking levels. In many instances, one can find many other intermediate checking levels.

We also need the notion of an automorphism group of a partial solution $(x_1, \ldots, x_{\alpha(k)})$. We have to be careful. The automorphism group of an object depends on the symmetry group acting on the object. We use the automorphism group in three different contexts, each with a different symmetry group. In the first context, the automorphism group is used to calculate the size of the orbit generated by $(x_1, \ldots, x_{\alpha(k)})$ under $\mathbf{S}_k(\mathscr{P})$. We denote this automorphism group by G_k, or more descriptively as $\langle x_1, \ldots, x_{\alpha(k)} | *^{n-\alpha(k)} \rangle$. It is a subgroup of $\mathbf{S}_k(\mathscr{P})$ fixing the partial solution $(x_1, \ldots, x_{\alpha(k)}, *, \ldots, *)$.

In the second context, we use the automorphism group of $(x_1, \ldots, x_{\alpha(k)})$ to partition the candidate vectors from $C_{\alpha(k)+1} \times \cdots \times C_{\alpha(k+1)}$ into orbits. This group is the subgroup of $\langle *^{\alpha(k)} | *^{\alpha(k+1)-\alpha(k)} | *^{n-\alpha(k+1)} \rangle$ which fixes the n-tuple $(x_1, \ldots, x_{\alpha(k)}, *, \ldots, *)$.

This automorphism group is denoted by G'_k or $\langle x_1, \ldots, x_{\alpha(k)} | *^{\alpha(k+1)-\alpha(k)} | *^{n-\alpha(k+1)} \rangle$. Since $G'_k = G_k \cap \langle *^{\alpha(k)} | *^{\alpha(k+1)-\alpha(k)} | *^{n-\alpha(k+1)} \rangle$, it can be found without performing extra isomorphism testing.

In the third context, we want to compute the orbit size of $(x_1, \ldots, x_{\alpha(k)})$ under G'_{k-1} of its parent. We denote this third version by G''_k or $\langle x_1, \ldots, x_{\alpha(k-1)} | x_{\alpha(k-1)+1} \ldots x_{\alpha(k)} | *^{n-\alpha(k)} \rangle$. The groups G_k and G''_k are defined for $k = 1, \ldots, N$. Whereas the group G'_k is defined for $k = 0, \ldots, N-1$.

For further discussion, we use the qualifiers *reduced* and *expanded*, respectively, to distinguish the search trees with and without isomorph rejection. The term *reduced* is used only if there is a chance of confusion, otherwise the unqualified term *search tree* means a *reduced search tree.*

3. Isomorph Rejection Algorithm

We assume the existence of an isomorphism testing routine which returns, for each partial solution $(x_1, \ldots, x_{\alpha(k)})$, a certificate which is unique and the same for every partial solution in the same orbit as $(x_1, \ldots, x_{\alpha(k)})$ under the action of a specific symmetry group. Moreover, we also assume that the routine returns the automorphism group of $(x_1, \ldots, x_{\alpha(k)})$.

During the running of the backtrack search with isomorph rejection, we have to maintain information about the certificates. With each certificate, we keep the following information.

1. An *expected_occurrence* field which is the size of the orbit generated by the certificate at testing level k under the action of $\mathbf{S}_k(\mathscr{P})$.
2. A *repetition_count* field which counts the number of images that have been accounted for.
3. A *first_incarnation* field which points to the first partial solution which gives rise to this certificate.

A certificate at a checking level is called a *checking* certificate. It is *active* if its images under the action of the symmetry group have not yet been completely accounted for. Otherwise, it is *inactive*. Of course, a checking certificate is active if, and only if, its repetition_count is smaller than its expected_occurrence. For a non-checking certificate, the final repetition_count is usually greater than its expected_occurrence. Thus, we have to use a different method to determine whether it is active. A non-checking certificate is *inactive* if one of its descendent checking certificates is inactive; otherwise, it is *active*. In order to save space, our method maintains only the active certificates. However, during debugging, it is advisable to keep the inactive certificates too, because if any of them is encountered again, then it is a clear indication of an error.

The fields in the declaration of a certificate should be expanded to maintain the list of certificates. This list should be maintained in a way to facilitate the operations of search, insertion and deletion. A height balanced binary search tree is one of the suitable data structures.

We assume that the backtrack search is done in a depth-first manner. When a partial solution $(x_1, \ldots, x_{\alpha(k)})$ is generated, its parent $(x_1, \ldots, x_{\alpha(k-1)})$ must be a certificate. Otherwise, we would not have generated $(x_1, \ldots, x_{\alpha(k)})$. However, it need not be active. Nevertheless, its automorphism groups are known, as a result of having survived the isomorphism testing.

Information about a partial solution is stored in a *node*. There is a *parent* link which points to its parent. The *eldest_child* link is used to point to the beginning of the list of its children. This list is maintained as a doubly linked list using the pointers *elder_sibling* and *younger_sibling*. The size of the orbit under the automorphism group of its parent is stored in the field *num_twin*. The link *isom* points to its certificate. All the partial solutions isomorphic to a given certificate are singly linked by the *next_incarnation* field. A node is said to be *active* or *inactive* according to whether its certificate is active or not. As in the case for a certificate, an inactive node does not have to be kept.

The isomorphism testing of the new partial solution at level k is divided into two parts. The first part tests whether it is an orbit representative under the action of the automorphism group G'_{k-1} of its parent. If it is not an orbit representative, it is thrown out. If it is, then a node is created for it and its orbit size is stored in the field *num_twin*. Each partial solution surviving this test is called a *pseudo-certificate*. There is a node for each pseudo-certificate.

In the second part of the isomorphism test, one applies the complete $S_k(\mathscr{P})$. The resulting certificate is tested against all the active certificates at testing level k. If it is new and active, then it is added to the certificate list. The expected_occurrence and the initial repetition_count are computed. If k is not the last testing level, then the group G'_k is also computed because it is required when extending the partial solution to the next testing level.

If the certificate is not new, then it is isomorphic to an old active partial solution. We update the repetition_counts of all the certificates in the branch of the search tree rooted at this old partial solution. These nodes must all be active; otherwise, we are seeing too many images of its certificate. We shall now discuss how to maintain the repetition counts.

We first consider the number of times that the partial solution $(x_1, \ldots, x_{\alpha(k)})$ occurs in the expanded search tree. Some of these occurrences are eliminated in the reduced search tree because a node is not an orbit representative under the action of the automorphism group G'_{k-1} of its parent. The number of eliminated occurrences is equal to the product of orbit sizes (num_twins) of nodes on the path from the root to the current partial solution inclusively. Suppose we let R denote the value of this product of orbit sizes. If $(x_1, \ldots, x_{\alpha(k)})$ is a new certificate, then its repetition_count is initialized to R. If it is not a new certificate, then from the first_incarnation field of the certificate, one can find the earliest partial solution $(y_1, \ldots, y_{\alpha(k)})$ which is isomorphic to $(x_1, \ldots, x_{\alpha(k)})$. The generation of $(x_1, \ldots, x_{\alpha(k)})$ is equivalent to finding $(y_1, \ldots, y_{\alpha(k)})$ R times. Moreover, if its descendant $(y_1, \ldots, y_{\alpha(j)})$ is a certificate, then its repetition_count should also be increased by R times the orbit sizes (num_twins) in the path from $(y_1, \ldots, y_{\alpha(k)})$ to $(y_1, \ldots, y_{\alpha(j)})$ but not including the one at $(y_1, \ldots, y_{\alpha(k)})$. This process of updating the repetition counts is summarized in the following procedure update_count.

```
Procedure update_count(certificate, R);

{recursive procedure to update the repetition_counts of the
 branch of the search tree rooted at the given certificate
 by R. If the certificate becomes inactive, it is pushed
 onto an inactive certificate stack for future processing.}
```

1. Increment the repetition_count of the certificate by R.
2. If the certificate is at a checking level and becomes inactive, push it onto the stack of certificates to be deleted.

3. For each child of the first occurrence of the certificate call update_count (child's certificate, R*child's orbit_size).

This updates all the repetition_counts. Then we delete all the inactive certificates that are stored on the stack. Essentially, we delete all the incarnations of the inactive certificates and the associated inactive nodes. Details of this process are given in the following procedure delete_certificate. One should note that if a node q becomes inactive, then its parent must be also inactive. Otherwise, either its parent will stay active forever, or when the repetition_count of its parent is incremented, q's repetition_count will be too large. In fact, if a link to the parent still exists and the parent is at a checking level, then the parent must be in the stack waiting to be deleted. This is a good early error detection test.

Procedure delete_certificate(certificate);
{procedure to delete all the incarnations of
the given inactive certificate.}

1. For each incarnation q of the certificate do steps 1.1 to 1.4.
1.1 Unlink q from its sibling's list.
1.2 Reset the parent links of all of q's children to nil.
1.3 If q's parent exists, push it onto the stack.
1.4 Dispose of q.
2. Dispose of the certificate.

As a conclusion to this section, we give a summary of the isomorph rejection test.

Given a partial solution $(x_1, \ldots, x_{\alpha(k)})$, we do:

1. If $(x_1, \ldots, x_{\alpha(k)})$ is not an orbit representative under the automorphism group G'_{k-1} of $(x_1, \ldots, x_{\alpha(k-1)})$ then exit.
2. Create a node for $(x_1, \ldots, x_{\alpha(k)})$.
3. Find its certificate and check it against the list of certificates at testing level k.
4. If it is not new, call update_count to update the repetition_counts. After all the repetition_counts are updated, call delete_certificate repeatedly to delete all the inactive certificates on the stack.
5. If the certificate is new, append it to the list and return an indication that $(x_1, \ldots, x_{\alpha(k)})$ should be further extended, if necessary.

4. Examples

We shall present two examples. In the first example, every testing level is a checking level. In the second example, only the first and last testing levels are checking levels.

In the first example, we shall consider generating all the 5×5 (0, 1)-matrices with exactly two ones in each row and column. Since the matrix has 25 entries, there are 25 normal levels and all the candidate sets C_i's are equal to $\{0, 1\}$. We shall generate the matrices row by row and apply isomorph rejection only when a row is completed. Thus, there are 5 testing levels and the corresponding candidate sets are the 10 possible ways of placing two ones in a row.

The symmetry group used is of size $(5!)^2$, consisting of all the combinations of independent row and column permutations. The groups $\mathbf{S}_k(\mathscr{P})$ for $k = 1$ to 5 have size $5!k!(5-k)!$, because the first k rows have to be permuted amongst themselves.

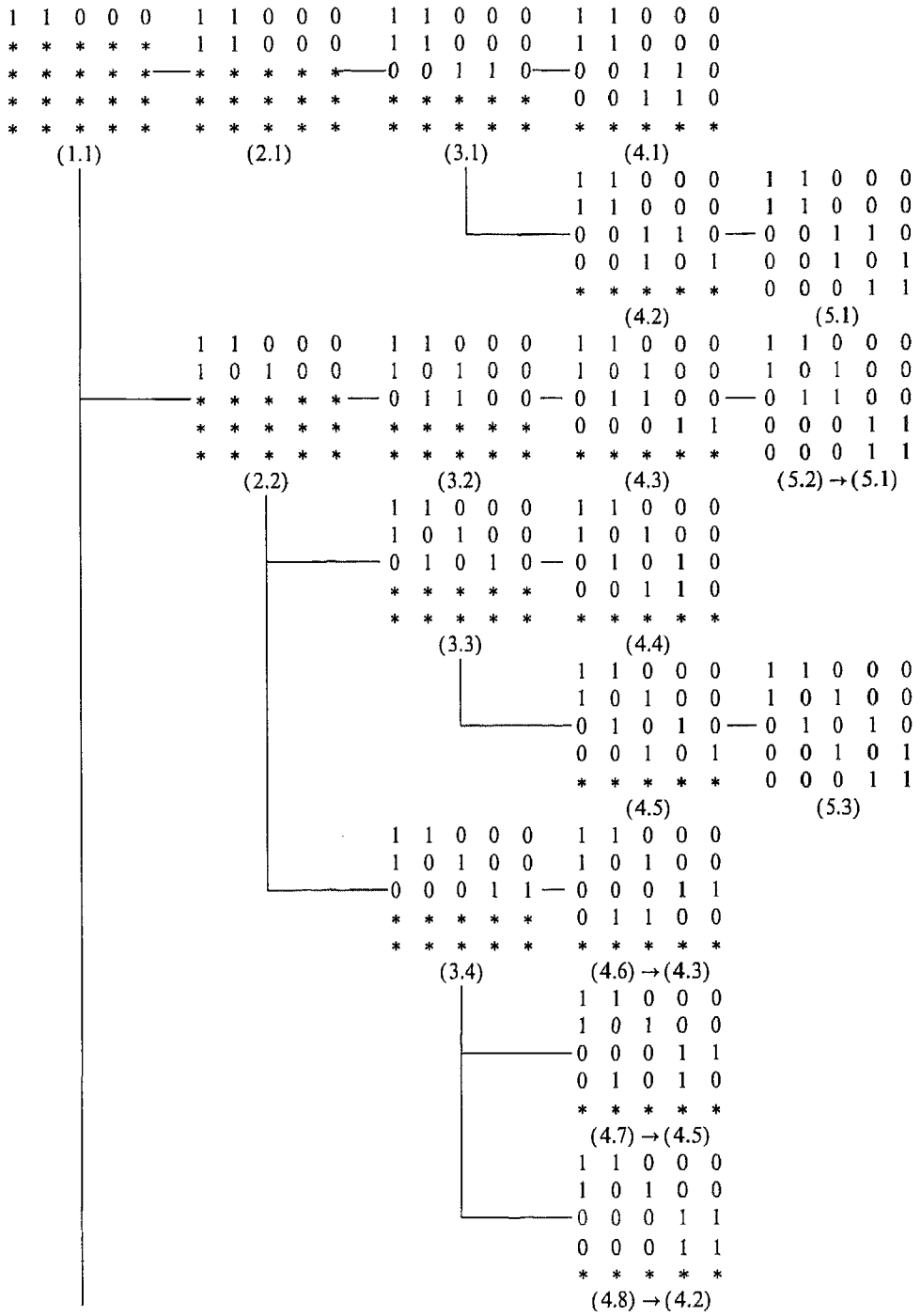

Fig. 1. Search tree for Example 1, Part 1.

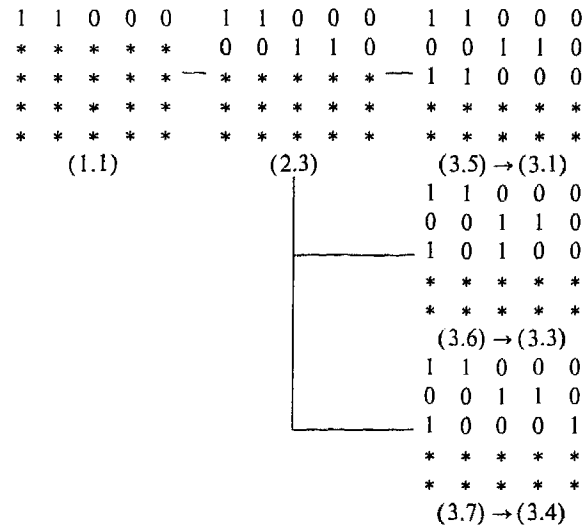

Fig. 2. Search tree for Example 1, Part 2.

In this example, every testing level is a checking level. Because of the independent nature of the row and column permutations, any permutation g that maps one partial solution at testing level k to another still leaves the permutations of the undefined rows free. We can choose to define the action of g on the undefined rows as being the identity permutation. This ensures that g is in $\mathbf{S}_j(\mathscr{P})$ for any $k \leqslant j$.

Figures 1 and 2 show the nodes in the reduced search tree. The nodes are labelled level by level from left to right. When there is a notation like (3.6) → (3.3), it means that the node which is labelled (3.6) is isomorphic to the node labelled (3.3). The *'s represent the wild card character. Figure 3 gives detailed information about the nodes. It shows whether the node is a certificate, the sizes of the three automorphism groups, G_k, G'_k and G''_k, the number of twins (num_twin), the expected number of occurrences (expected_occurrence), and the initial value of the repetition_count.

Let us consider some selected nodes on the search tree. At the first testing level, G'_0 is the same group as $\mathbf{S}_1(\mathscr{P})$. The group G_1 is the same as G''_1 and it is a direct product of permuting the first 2 columns, permuting the last 3 columns and permuting the bottom 4 rows. In the group G'_1, the second row is distinguished. Thus, its size is smaller by a factor of 4. The field num_twin is computed by $|G'_0|/|G''_1| = 2880/288 = 10$. The expected_occurrence is equal to $|\mathbf{S}_1(\mathscr{P})|/|G_1| = 2880/288 = 10$. The initial repetition_count is the same as num_twin and is also 10. Since the last two values are the same, it is inactive as soon as it is created.

As for the node (2.1), all its 3 automorphism groups are different. As a subgroup of $\mathbf{S}_2(\mathscr{P})$, G_2 consists of independent permutations of the first 2 rows, the first 2 columns, the last 3 rows and the last 3 columns. Thus, $|G_2| = 144$. As a subgroup of G'_1, one cannot permute the first two rows. Hence, $|G''_2|$ is only 72. As for G'_2, row 3 is distinguished. Thus, one can only permute the last two rows. Hence, $|G'_2|$ is smaller than $|G_2|$ by a factor of 3. Since the initial repetition_count is equal to the expected_occurrence, it is also inactive as soon as it is created.

The first active node is (3.1). It only becomes inactive when (3.5) is generated. For the

node	parent	certif.	auto G_k''	group G_k	size G_k'	num twin	expected occur.	repetit. count
1.1	—	yes	288	288	72	10	10	10
2.1	1.1	yes	72	144	48	1	10	10
2.2	1.1	yes	12	24	8	6	60	60
2.3	1.1	yes	24	48	16	3	30	30
3.1	2.1	yes	16	16	8	3	90	30
3.2	2.2	yes	8	24	12	1	60	60
3.3	2.2	yes	2	4	2	4	360	240
3.4	2.2	yes	8	8	4	1	180	60
3.5	2.3	no	8	16	—	2	—	—
3.6	2.3	no	4	4	—	4	—	—
3.7	2.3	no	4	8	—	4	—	—
4.1	3.1	yes	8	32	32	1	90	30
4.2	3.1	yes	4	8	8	2	360	60
4.3	3.2	yes	12	12	12	1	240	60
4.4	3.3	yes	2	8	8	1	360	240
4.5	3.3	yes	1	2	2	2	1440	480
4.6	3.4	no	4	12	—	1	—	—
4.7	3.4	no	1	2	—	4	—	—
4.8	3.4	no	4	8	—	1	—	—
5.1	4.2	yes	8	24	—	1	600	60
5.2	4.3	no	12	24	—	1	—	—
5.3	4.5	yes	2	10	—	1	1440	480

Fig. 3. Information on the nodes (Example 1).

node (3.5), the product of the num_twin fields from (1.1) to (3.5) is $10 \cdot 3 \cdot 2 = 60$. Thus, it increases the repetition_count of (3.1) from 30 to 90, making it inactive.

Before the generation of the last node (3.7), there are still several active nodes. However, the mapping from (3.7) to (3.4) implies that nodes (4.6) to (4.8) are rediscovered, which makes the nodes (4.2), (4.3) and (4.5) inactive. The rediscovery of (4.3) and (4.5) also makes (5.1) and (5.3) inactive. One should note also that the contribution from (3.7) to (5.3) is $10 \cdot 3 \cdot 4 \cdot 4 \cdot 1 = 480$. The second "4" in the product is from the num-twin field of the node (4.7).

The nodes (4.1) and (4.4) both have no successors at testing level 5. Yet, they are certificates. They stay active until nodes (3.5) and (3.6) are generated.

From this example, one sees that a certificate such as (5.1) receives contributions towards its repetition_count from many different parts of the search tree. It is unlikely that the counts are correct if some branches are pruned erroneously, either by a software bug or a hardware malfunction.

```
0 * * *    0 $ $ $    0 $ $ $
* 0 * *    $ 0 * *    $ 0 $ $
* * 0 *    $ * 0 *    $ $ 0 *
* * * 0    $ * * 0    $ $ * 0
  (i)        (ii)       (iii)
```

Fig. 4. Templates for partial solutions (Example 2).

We shall now consider another example, where only the first and the last testing levels are checking levels. The problem is the generation of all regular graphs of degree 2 on 4 vertices. We shall generate the adjacency matrices of these graphs. Because the adjacency matrix is symmetric, we only have to generate the 6 entries above the diagonal. Thus, there are 6 normal levels. Two matrices define the same graph if one can be changed into the other by simultaneous row and column permutations. We shall generate the matrices row by row and apply isomorph rejection only when a row is completed. Thus, there are only 3 testing levels.

Let us first work out the symmetry groups. Figure 4 gives the templates of the partial solutions at every testing level. In these templates, we use an extra undefined symbol '$' in order to clarify the action of the symmetry group, by insisting that a '$' has to be mapped to a '$' and a '*' to a '*'. The group $\mathbf{S}(\mathscr{P})$ fixes the configuration (*i*). It contains all the 4! simultaneous row and column permutations. The group $\mathbf{S}_1(\mathscr{P})$ fixes the configuration (*ii*). It contains the 3! ways of simultaneously permuting the last 3 rows and columns. The group $\mathbf{S}_2(\mathscr{P})$ fixes the configuration (*iii*). It contains only 4 elements, because the columns 3 and 4 (simultaneously, rows 3 and 4) form a separate orbit from columns 1 and 2.

Figure 5 shows the search tree and Fig. 6 gives the detailed information about the nodes. The node (2.1) is a dead end. One notes that for both the nodes (2.1) and (2.2), the expected_occurrences are 2, but the initial repetition_counts are already 3. One may wonder why the repetition_count is larger than the expected_occurrence. The expected_occurrence of node (2.1) gives the number of its images at testing level 2. The node (1.1) has 3 images in testing level 1. Two of these images have extensions at testing level 2 which are isomorphic to (2.1). The third image obtained by simultaneous permutation of rows 2 and 4 and columns 2 and 4, does not have an extension at testing level 2 which is isomorphic to (2.1). In fact, if we apply the inverse permutation to (2.1), we get an object with the symbol '*' at the second row. Such an object is not generated and hence

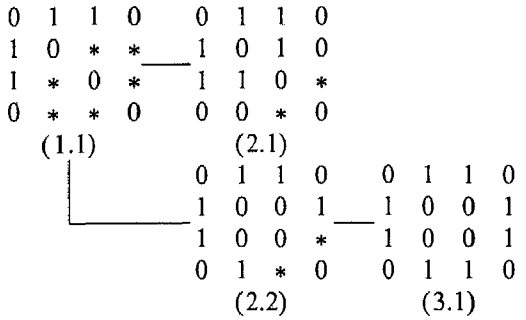

Fig. 5. Search tree for Example 2.

node	parent	certif.	auto G_k''	group G_k	size G_k'	num twin	expected occur.	repetit. count
1.1	—	yes	2	2	1	3	3	3
2.1	1.1	yes	1	2	2	1	2	3
2.2	1.1	yes	1	2	2	1	2	3
3.1	2.2	yes	2	8	—	1	3	3

Fig. 6. Node information for Example 2.

inflates the count. This is an example of the general behaviour of a non-checking level. The repetition_count is different from the expected_occurrence.

5. Concluding Remarks

Let us first prove that the proposed algorithm works. The proof is based on the concept that if more information about the permutations is kept, then one can actually construct the mappings that take a certificate to all its isomorphic copies. Suppose we make the following changes:

1. For each node at testing level k in the search tree, we keep a set $U(k)$ of coset representatives of G_k'' in G_{k-1}'. Of course, the size of $U(k)$ is exactly equal to num_twin.
2. If a node is not a certificate, then, in addition to a pointer to its certificate, we also keep a permutation g_k which maps the node to the earliest isomorphic partial solution.
3. For each certificate at testing level k, we keep a set of *words* consisting of products of elements from the sets defined in steps 1 and 2. The size of this set is always equal to the value of repetition_count. When the certificate is first created, this set is initialized to

$$\left\{\prod_{i=1}^{k} u(1)u(2)\ldots u(k),\ u(i)\in U(i)\right\},$$

where the $U(i)$'s are for the nodes on the path from the root to the certificate. The procedure update_count is also changed so that is passes an additional parameter which is a set of words. At the initial call to update_count for a node at testing level k, this set is initialized to

$$\left\{\prod_{i=1}^{k} u(1)u(2)\ldots u(k)g_k,\ u(i)\in U(i)\right\},$$

where the $U(i)$'s are for the nodes on the path from the root to the node and g_k maps the node to its earliest isomorphic partial solution. This set is added to the set of words kept for the certificate. For each subsequent recursive call to update the counts of a child node at testing level j, this parameter set is updated by multiplying all its elements on the right by $u(j)g_j$ where $u(j)\in U(j)$ for the child, and g_j maps

the child to its isomorphic partial solution. Thus, the size of this set is always equal to the value of R, the increment for the repetition_count.

We shall prove that for each checking certificate at checking level k, the set of words is a set of coset representatives for G_k in $\mathbf{S}_k(\mathscr{P})$. We prove it by constructing a 1-1 onto mapping from the set of isomorphic partial solutions to the set of words. Let B be one of these partial solutions. We also let $q|_i = (x_1, \ldots, x_{\alpha(i)})$ denote the first i components of the n-tuple q used in the following description.

1. Initial $p =$ identity and $q = B = (x_1, \ldots, x_{\alpha(k)})$.
2. For each testing level i from 1 to k do steps 2.1 to 2.5.
2.1 Let $u(i)$ map $(x_1, \ldots, x_{\alpha(i)}, *, \ldots, *)$ to its orbit representative under G'_{i-1}.
2.2 $q := q^{u(i)}$.
2.3 Let g_i map $q|_i$ to its certificate under $\mathbf{S}_i(\mathscr{P})$.
2.4 $p := p \cdot u(i) \cdot g_i$.
2.5 $q := q^{g_i}$,

We claim that on entry to the ith iteration of the above loop, the partial solution $q|_{i-1}$ is a certificate in the reduced search tree. This claim is vacuously true when i is 1. Suppose it is true on the ith iteration. It then makes sense to talk about mapping $q|_i$ to its orbit representation under G'_{i-1}. At the end of the ith iteration, the mapping $u(i)g_i$ has been applied to the original partial solution $q|_i$ to ensure that the new one is a certificate and hence in the search tree.

At the end of the kth iteration, p is a word in the set of words for the certificate and it maps B to its certificate. The construction process is well defined. Hence, for each B, there corresponds a p. Distinct B's lead to distinct words. Moreover, if k is a checking level, then every word is a product of elements in $\mathbf{S}_k(\mathscr{P})$. Thus, the inverse of every word maps the certificate to a partial solution at testing level k. Hence, the mapping from the set of isomorphic partial solutions to the set of words is 1-1 and onto. Since the size of the set of words is the increment for the repetition_count, the final value of the repetition_count must be equal to the expected_occurrence.

There is a price to be paid for having the consistency check. In the first example of the previous section, one should not have to consider the node (2.3). Since there are two 1's in the first column, by a suitable row permutation, we can guarantee that the second row starts with a 1. This kind of reasoning is typical of a "pigeon hole" analysis that implies a particular starting position can be assumed. It reduces the choices at the early levels. If one uses the method in this paper, one has to look at more starting positions. Fortunately, these extra branches are usually small, because the isomorph rejection will sooner or later catch up and delete all its descendants. When comparing the extra work versus the comfort of having an independent consistency check, we tend to choose the latter.

Besides catching real bugs, the consistency check will sometimes fail on a program that "works", in the sense that it does find all the non-isomorphic solutions. There are several possible reasons. One common source of problems is because the program has taken shortcuts in evaluating the predicate. When it is computationally too expensive to determine the actual value of a predicate, a true value is returned because it will not lead to a loss of solutions. Whereas, for some of its images, it may be easy to determine that the predicate is false. Then the counts will not agree.

Another common source of problems is that a wrong symmetry group is used. If one uses more symmetry than there actually is, then it is of course a real bug. To be on the safe side, one often uses a smaller or even an incomplete group. This confusion in the symmetry group often leads to counts that do not agree. Locating the source of the problem is often time consuming but rewarding, because one often identifies more symmetry operations which lead to a smaller search tree. Besides, one is left with a "good" feeling that one finally "understands" the program.

References

Butler, G., Lam, C. W. H. (1985). A general backtrack algorithm for the isomorphism problem of combinatorial objects. *J. Symb. Comp.* **1**, 363–381.

Golomb, S. W, Baumert, L. D. (1965). Backtrack programming. *J. Comb. Theory* **4**, 516–524.

Hall, M. Jr. (1959). *The Theory of Groups.* New York: Macmillan.

Lam, C. W. H., Thiel, L., Swiercz, S., McKay, J. (1983). The nonexistence of ovals in a projective plane of order 10. *Discrete Math.* **45**, 319–321.

Lam, C. W. H., Thiel, L., Swiercz, S. (1986). The nonexistence of code words of weight 16 in a projective plane of order 10. *J. Comb Theory. Ser. A*, **42**, 207–214.

Purdom, P. W. Jr., Brown, C. A. (1985). *The Analysis of Algorithms.* New York: Holt, Reinhart and Winston.

Reingold, E. M., Nievergelt J., Deo, N. (1977). *Combinatorial Algorithms: Theory and Practice.* Englewood Cliffs, NJ: Prentice-Hall.

Swift, J. D. (1958). Isomorph rejection in exhaustive search techniques. *Proc. AMS Symp. Appl. Math* **X**, 195–200.

Walker, R. J., (1960). An enumerative technique for a class of combinatorial problems. *Proc. AMS Symp. Appl. Math.* **X**, 91–94.